UNIVERSITÀ DI PISA

Master's degree in Computer Engineering

598II Electronic Systems

UART Transmitter

Designer:

Gabriele Suma

ACADEMIC YEAR 2023/2024

Contents

Chapter 1

Introduction

1.1 Overview

A **Universal Asynchronous Receiver-Transmitter** is a computer hardware device for asynchronous serial communication in which the data format and transmission speeds are configurable. It sends data bits one by one framed by start and stop bits so that precise timing is handled by the communication channel. In particular, it can send words using different protocol configuration, in terms of:

- **Number of data bits**
- **Baud Rate**
- **Parity**
- **Number of start and stop bits**

The task assigned required the design of a digital system in order to meet the following specifications:

- Number of data Bits: **7**
- Baud Rate: **115200**
- Parity: **even**
- Number of start and stop bits: **2**

Chapter 2

Description

2.1 Interface design

The interface of the circuit has to be designed as it follows [Figure 2.1]:

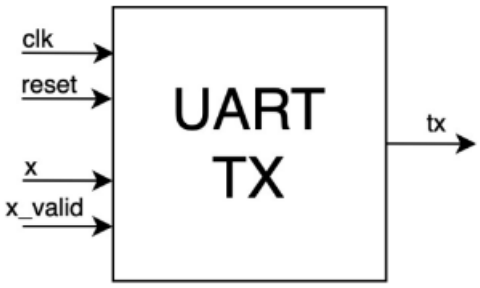


Figure 2.1: UART Transmitter interface

2.2 Circuit description

In the project's initial phase, understanding the potential architectures for transmitter implementation was essential. After identifying and understanding the necessary component for developing the UART's functionalities, the following block diagram was produced [Figure 2.3]:

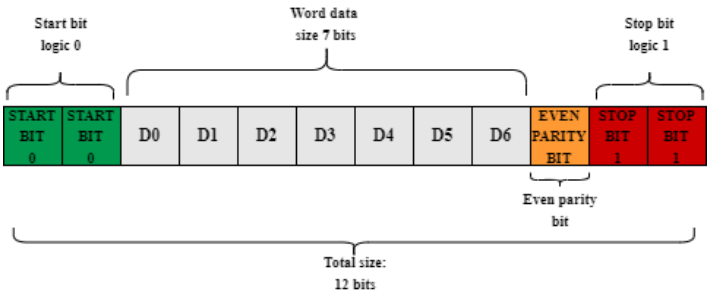


Figure 2.2: Word sent using N = 7, parity even and number of start/stop bits = 2.

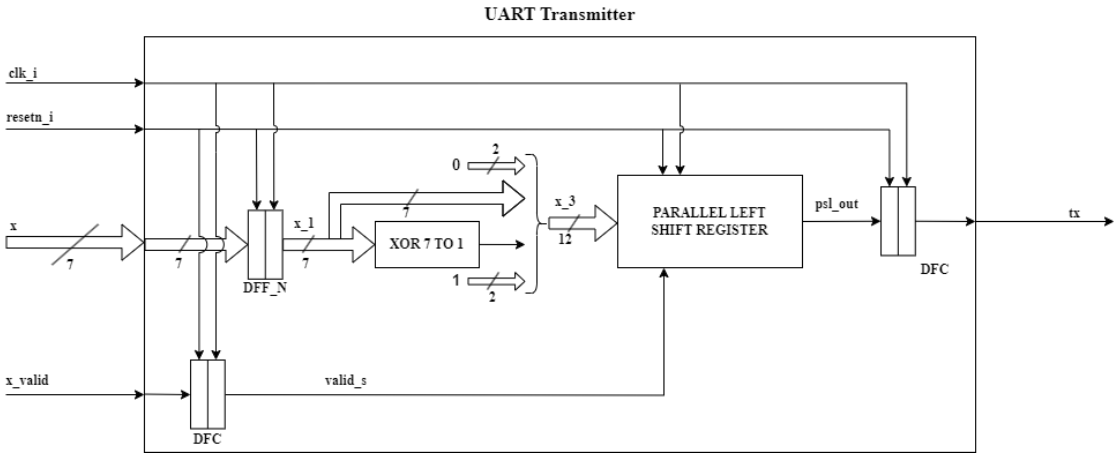


Figure 2.3: UART Transmitter block diagram

- **Inputs:** clk_i, resetn_i, x, x_valid
- **Output:** tx

Assumptions

- The input **x** is interpreted as the word transmitted by a protocol. In this scenario, the UART transmitter receives the complete **7-bit word**.
- The remaining inputs and the output **tx** are interpreted on a **single bit**.

From the block diagram, it is quite clear that it was necessary to use multiple components:

1. **DFC**: This is a standard **flip-flop** capable of storing a single bit.
2. **DFF_N**: This is a standard **flip-flop** capable of storing N bit (In UART's case $\mathbf{N = 7}$).
3. **XOR 7 TO 1**: This represents an **XOR circuit** that accepts a 7-bits input and delivers a **single-bit output**.
4. **Parallel left shift register**: This is an enhanced version of the basic **left shift register**. It is designed to accept a **12-bit input** and produce a **single-bit output**.

Chapter 3

VHDL Implementation

3.1 Principle of working

Based on the components outlined above [section 2.2], when a new word is sent to the UART transmitter [Figure 2.3], the latter **functions** according to the following **sequence**:

1. The new **word** is stored in the **DFF_N**.
2. The word, once stored in the flip-flop, is fed into the **XOR circuit** that computes the **even parity bit** needed for transmission.
3. After performing the XOR computation, the **initial word** is combined with its corresponding start-stop bits and its parity bit. This combination results in a total of **12 bits**. Subsequently, this 12-bit data is stored into the **parallel shift left register**.
4. The **parallel shift left register** has the responsibility of transmitting the single-bit, adhering to the constraints imposed by the **baud rate**.
5. Eventually, the last single-bit flip-flop stores the parallel left shift register output and it will be transmitted to the external world.

Essentially, to respect the constraints set forth by [item 4], it was necessary to clearly comprehend what is the **baud rate**. Baud rate denotes the quantity of signal alterations occurring every second. In this instance, with a rate of **115200**, each bit has a duration of **8.681 µs**, which requires **1086 clock cycles** implementing a clock period of **8 ns / 125 MHz**.

$$ClksPerBit = \frac{Bit-Duration}{Clock-Period} = \frac{8681 \text{ ns}}{8 \text{ ns}} = \mathbf{1086}\ cycles$$

3.2 XOR N to 1

In order to compute the **even parity-bit** for each word received by the UART transmitter, the properties of the **XOR gate** were effectively utilized. In this particular case, a **7 to 1 XOR** was implemented. However, the written code is designed to be compatible with any other protocol that may be required. Specifically, it was achieved this by cascading **six XOR gates**, as depicted in the referenced [Figure 3.1].

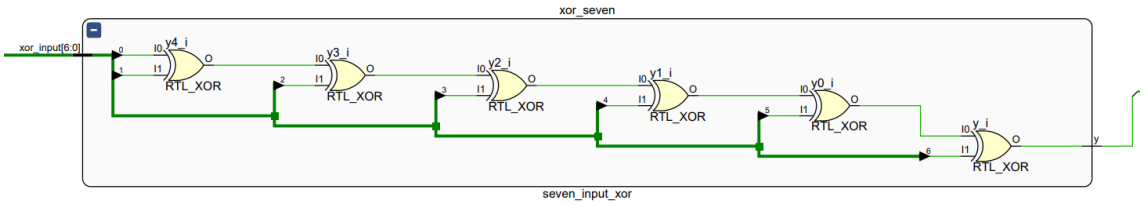


Figure 3.1: XOR with 7 inputs and 1 output.

3.2.1 VHDL code

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity seven_input_xor is
5      generic(
6          N : integer := 7
7      );
8      port(
9          xor_input : in std_logic_vector(N-1 downto 0);
10         y : out std_logic
11     );
12 end entity seven_input_xor;
13
14 architecture xor_arch of seven_input_xor is
15 begin
16     p: process(xor_input)
17     variable xor_result : std_logic;
18     begin
19         xor_result := xor_input(0);
20         for i in 1 to N-1 loop
21             xor_result := xor_result xor xor_input(i);
22         end loop;
23     y <= xor_result;
24     end process;
25 end architecture;
```

Listing 3.1: Code for xorNto1.vhd

3.3 Parallel shift left register

This component, which serves as the **core** of the system, implements the **majority** of the UART functionalities. Throughout the design process, I successfully implemented two VHDL code version to define the parallel shift left register behaviour.

3.3.1 First version

The initial version was designed using a **single sequential process** with asynchronous reset. Here is the code [Listing 3.2]:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity shift_left_register is
5      generic (
6          ClksPerBit : positive := 1086;
7          -- This is the number of clock cycles per bit transmitter with Baud
       Rate 115200. Essentially it can be adjusted if the Baud Rate is
      changed.
8          Nbit: positive := 12
9          -- Due to others protocols, the number of bits can be changed.
10     );
11     port(
12         clk : in std_logic;
13         reset : in std_logic;
14         load : in std_logic;
15         data_in : in std_logic_vector(Nbit-1 downto 0);
16         shift_out : out std_logic
17     );
18 end shift_left_register;
19
20 architecture behavior of shift_left_register is
21     signal reg : std_logic_vector(Nbit-1 downto 0) := (others => '0');
22      -- A signal used to correctly shift the data.
23     signal clock_counter : integer range 0 to ClksPerBit-1 := 0;
24     -- Counter needed to count the clock cycles.
25     signal idle : std_logic := '1';
26     -- A signal used to indicate that the data is not being transmitted.
27     signal count : integer range 0 to Nbit+1 := 0;
28     -- A counter used to count the number of bits transmitted.
29 begin
30
31     process(clk, reset)
32
33     begin
34         --We are in the reset state.
35         if reset = '1' then
36         -- all the signals are set to 0 in order to avoid latches
37             reg <= (others => '0');
38             clock_counter <= 0;
39             count <= 0;
40         elsif rising_edge(clk) then
41         -- We are in the rising edge of the clock.
42             if load = '1' and count = 0 then
43             -- If the load signal is 1, the data is loaded into the
       register.
44                 reg <= data_in;
45                 -- The data is loaded into the register.
46                 idle <= '0';
```

```
47                  -- The idle signal is set to 0 in order to indicate that
    the system is busy.
48                  clock_counter <= 0;
49                  -- The clock counter is set to 0 in order to start counting
     the clock cycles.
50                  count <= 1;
51                  -- The count is set to 1 in order to start counting the
    bits.
52              elsif clock_counter = ClksPerBit-1 and  count > 0 and count <
    Nbit  then
53              -- If the clock counter is equal to the number of clock cycles
    per bit
54                  reg <= reg(Nbit-2 downto 0) & '1';
55                  -- The data is shifted to the left.
56                  clock_counter <= 0;
57                  -- The clock counter is set to 0 in order to start counting
     the next clock cycles.
58                  idle <= '0';
59                  -- The idle signal is set to 0 in order to indicate that
    the system is busy.
60                  count <= count+1;
61                  -- The count is incremented by 1 because a bit has been
    transmitted.
62              elsif count = Nbit and clock_counter = ClksPerBit-1 then
63              -- If the count is equal to 12 and the clock counter is equal
    to the number of clock cycles per bit
64                  count <= 0;
65                  -- All the bits have been transmitted, so the count is set
    to 0.
66              elsif count /= 0 then
67              -- If the count is not equal to 0
68                  clock_counter <= clock_counter+1;
69                  -- Increment the clock counter by 1.
70                  idle <= '0';
71                  -- The idle signal is set to 0 in order to indicate that
    the system is busy.
72              else
73                  idle <= '1';
74                  -- The idle signal is set to 1 in order to indicate that
    the system is idle.
75              end if;
76          end if;
77      end process;
78      shift_out <=idle or reg(Nbit-1);
79      -- The shift_out signal is set to 1 if the system is idle, otherwise it
     is set to the MSB of the register.
80  end behavior;
```

Listing 3.2: First version of the parallel shift left register.

3.3.2 Second version

On the other hand, **the final version** was designed using two processes: a **sequential** and a **combinational** one. The second process was specifically implemented to manage each **state of the system**. Here is the code [Listing 3.3]:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
```

```
4 --
      ----------------------------------------------------------------------------------
5   -- Parallel shift left register with load on 12 bits input.
6   --
      ----------------------------------------------------------------------------------
7 entity shift_left_register is
8     generic (
9         Nbit: positive := 12;
10        ClksPerBit : positive := 1086
11        -- This is the number of clock cycles per bit transmitter with Baud
      Rate 115200.
12        --Essentially it can be adjusted if the Baud Rate is changed.
13
14    );
15    port(
16        clk : in std_logic;
17        reset : in std_logic;
18        load : in std_logic;
19        data_in : in std_logic_vector(Nbit-1 downto 0);
20        shift_out : out std_logic
21    );
22 end shift_left_register;
23
24 architecture behavior of shift_left_register is
25    type state is (S0,S1,S2); -- Definition of the state machine
26    signal current_state : state; -- It will be used a signal named
      current_state to store the current state of the state machine.
27    signal reg : std_logic_vector(Nbit-1 downto 0) := (others => '0'); --
      It will be used a signal named reg to store the data in the actual
      shift_left_register.
28    signal clock_counter : integer range 0 to ClksPerBit-1 := 1; -- It will
       be used a signal named clock_counter to count the number of clock
      cycles.
29    signal count : integer range 0 to 12 := 0; -- It will be used a signal
      named count to count the number of bits transmitted.
30 begin
31
32    process(clk, reset)
33    begin
34        if reset = '1' then
35        -- The reset is asynchronous.
36            current_state <= S0;
37             -- The state machine is reset.
38            reg <= (others => '0');
39            -- The register is reset.
40            clock_counter <= ClksPerBit-1;
41            -- The clock counter is reset.
42            count <= 0;
43            -- The counter is reset.
44        elsif rising_edge(clk) then
45           case current_state is
46             -- The state machine is implemented.
47            when S0 =>
48            if load = '1' then
49            -- When the load is high, the data_in is loaded into the
      register.
50                current_state <= S1;
51                -- The state machine goes into the next state.
52                reg <= data_in;
53                -- The data_in is loaded into the register.
54                clock_counter <= ClksPerBit-2;
```

```
55                  -- The clock counter is reset, but with a value of 1083,
    because we're loading shift_out with the first bit of the parallel
    shift left register.
56                  count <= 1;
57                  -- Count is set to 1, because we're loading shift_out with
    the first bit of the parallel shift left register.
58              end if;
59              when S1 =>
60                  if clock_counter = 0 then
61                  -- We wait for the clock_counter to reach 0, we need to
    wait 1085 clock cycles to transmit a bit.
62                      current_state <= S2;
63                      -- The state machine goes into the next state.
64                      clock_counter <= ClksPerBit-2;
65                      -- The clock counter is reset, but with a value of
    1083, because we're loading shift_out with the first bit of the
    parallel shift left register.
66                  else
67                      clock_counter <= clock_counter - 1;
68                      -- The clock counter is decremented, because we're
    waiting for the clock_counter to reach 0.
69                  end if;
70              when S2 =>
71                  if count = 12 then
72                  -- When the count reaches 12, we have transmitted all the
    bits.
73                      current_state <= S0;
74                      -- The state machine is reset.
75                  else
76                      reg <= reg(Nbit-2 downto 0) & '1';
77                      -- The parallel shift left register is shifted left.
78                      count <= count+1;
79                      -- The counter is incremented, because we have
    transmitted a bit.
80                      current_state <= S1;
81                      -- The state machine goes to the next state.
82                  end if;
83              end case;
84          end if;
85      end process;
86
87      p_OUTPUT_LOGIC: process(current_state, reg)
88      begin
89          shift_out <= '1';
90          -- The shift_out is set to '1' by default, in order to avoid
    latches.
91          case current_state is
92              -- Defining the output logic of the shift_out based on the
    current state of the state machine.
93              when S0 =>
94                  shift_out <= '1';
95              when S1 =>
96                  shift_out <= reg(Nbit-1);
97              when S2 =>
98                  shift_out <= reg(Nbit-1);
99          end case;
100     end process;
101 end behavior;
```

Listing 3.3: Second version of the parallel shift left register.

In the following sections of this report, I will discuss which of the **two previous designs** is the **most effective** for implementing the requested UART.

3.4 UART

The UART architecture is composed by **two single-bit flip-flops**, **one N-bits flip-flop**, **one xorNto1** and **one parallel shift left register**. The code below defines this architecture following the previously defined block diagram [Figure 2.3]:

```
1  library ieee;
2    use ieee.std_logic_1164.all;
3    use ieee.numeric_std.all;
4    use ieee.std_logic_misc.all;
5
6  entity uart is
7    generic(
8      Nbit : positive := 7; -- number of bits in the input
9      Mbit : positive := 12 -- number of bits for the parallel shifter
10 );
11
12   port(
13     clk_i : in std_logic;
14     resetn_i : in std_logic;
15     x : in std_logic_vector(Nbit-1 downto 0);
16     x_valid : in std_logic;
17     tx: out std_logic
18   );
19 end entity;
20
21 architecture rtl of uart is
22     -- x_1 is the output of the first flip flop and it is in input of the
       xorNto1
23     signal x_1 : std_logic_vector(Nbit-1 downto 0) := (others => '0');
24     -- x_3 is the signal that is loaded in the parallel shifter
25     signal x_3 : std_logic_vector(Mbit-1 downto 0) := (others => '0');
26     -- xor_out is the signal that is the output of the xorNto1
27     signal xor_out : std_logic := '0';
28     -- valid_s is the signal that is the output of the second flip flop
       which store the x_valid value
29     signal valid_s : std_logic := '0';
30     -- psl_out signal is the output of the parallel shifter
31     signal psl_out : std_logic;
32
33     component dff_n is --Flip flop with N bits
34     generic(
35         Nbit : positive := 7
36     );
37     port(
38       clk_i : in std_logic;
39       resetn_i : in std_logic;
40       di : in std_logic_vector(Nbit-1 downto 0);
41       en : in std_logic;
42       do : out std_logic_vector(Nbit-1 downto 0)
43     );
44     end component;
45
46     component DFC is --Flip flop with 1 bit
47         port(
48             clk_i : in std_logic;
49             resetn_i :  in std_logic;
50             d : in std_logic;
51             q : out std_logic
52         );
53     end component;
54
```

```
55      component shift_left_register is
56      -- Shift register with parallel loading
57
58      generic (
59          Nbit: positive := 12
60      );
61
62      port(
63          clk : in std_logic;
64          reset : in std_logic;
65          load : in std_logic;
66          data_in : in std_logic_vector(Nbit-1 downto 0);
67          shift_out : out std_logic
68      );
69      end component;
70
71      -- XOR with 7 inputs and 1 output
72      component seven_input_xor is
73          port(
74              xor_input : in std_logic_vector(Nbit-1 downto 0);
75              y : out std_logic
76          );
77      end component;
78
79    begin
80      --Instantiate the components
81      ff1: dff_n
82      generic map(
83          Nbit => 7
84      )
85      port map(
86          clk_i => clk_i,
87          resetn_i => resetn_i,
88          di => x, -- mapping the input x to the input of the first flip flop
89          en => '1',
90          do => x_1 -- mapping the output of the first flip flop to the input
        of the xorNto1
91      );
92
93
94      ff2: DFC
95      port map(
96          clk_i => clk_i,
97          resetn_i => resetn_i,
98          d => x_valid, -- mapping the input x_valid to the input of the
      second flip flop
99          q => valid_s -- mapping the output of the second flip flop to the
      input of the parallel shifter
100     );
101
102     parallel_shifter: shift_left_register
103     generic map(
104         Nbit => 12
105     )
106     port map(
107         clk => clk_i,
108         reset => resetn_i,
109         load => valid_s, -- useful to load the parallel shifter only when
      the input x_valid is asserted
110         data_in => x_3,  -- mapping the input x_3 to the input of the
      parallel shifter
111         shift_out => psl_out -- mapping the output of the parallel shifter
      to the input of the third flip flop
112     );
```

```
113
114     ff3: DFC
115     port map(
116         clk_i => clk_i,
117         resetn_i => resetn_i,
118         d => psl_out, -- mapping the output of the parallel shifter to the
    input of the third flip flop
119         q => tx -- mapping the output of the third flip flop to the output
    tx
120     );
121     -- combining all the start/stop bits and the parity bit with the input
    x
122     x_3(11 downto 10) <= (others =>'0');
123     x_3(1 downto 0) <= (others =>'1');
124     x_3(2) <= xor_out;
125     x_3(9 downto 3) <= x_1;
126
127     xor_seven: seven_input_xor
128     port map(
129         xor_input => x_1,
130         y => xor_out
131     );
132
133
134 end architecture;
```

Listing 3.4: UART code

3.5 Test-benches

In order to verify the system's functionality, three test-benches were created. All the test benches have been tested on both versions of the parallel shift left register. For convenience, the waveform figures are taken from the **second version**.

3.5.1 Test-bench n°1

The following code [Listing 3.5], partially mentioned for convenience, tests if the circuit behaves as expected. Essentially, it shows how the system reacts when the inputs are driven correctly.

```
1  STIMULUS: process
2          begin
3              -- Apply reset
4              resetn_i <= '1';
5              wait for clk_period;
6              resetn_i <= '0';
7              wait for clk_period;
8              -- Simulating two words sent respecting timing constraints. At
    the end it will be driven a reset.
9              -- Correctly driving both inputs in order to test the correct
    behaviour of the circuit.
10             x_valid <= '1';
11             x <= "1000000";
12             wait for clk_period;
13             -- Driving correctly x_valid to start the transmission.
```

```
14            x_valid <= '0';
15            wait for clk_period*clk_per_bit*12;
16            -- Here the circuit has just finished the transmission of the
    first word and it's going to start the transmission of the second one.
17            x_valid <= '1';
18            x <= "1111111";
19            wait for clk_period;
20            x_valid <= '0';
21            wait for clk_period*clk_per_bit*13;
22            --In this case it's going to wait another cycle in order to
    test if the circuit drives tx to '1' when the latter is idle.
23            resetn_i <= '1';
24            wait for clk_period;
25            resetn_i <= '0';
26            wait for clk_period;
27            x_valid <= '1';
28            x <= "0100010";
29            wait for clk_period;
30            x_valid <= '0';
31            wait for clk_period*clk_per_bit*12;
32            -- End of simulation
33            testing <= false;
34            wait;
35        end process;
```

Listing 3.5: First test-bench code.

Variable values:

- **clk_period** = 8 ns
- **clk_per_bit** = 1086

The code provided above represents the following flow of **inputs**:

1. Firstly, a **reset** signal is applied for a total of **clk_period*clk_per_bit ns**.
2. **x_valid** is set to **1** and **x** is set to **1000000** for a **clk_period ns**.
3. **x_valid** is set to **0** in order to start the transmission, the total transmission time is $T_{Transmission} = clk\_period * clk\_per\_bit * 12$.
4. Once $T_{Transmission}$ has elapsed, **x** is set to **1111111** and **x_valid** is driven correctly.
5. In this scenario, the delay before accepting a new input is extended by **clk_period*clk_per_bit ns**. This is done to verify the expected behavior of the UART during the idle state.
6. A reset signal is applied for a total of a **clk_period ns**.
7. Lastly, a new input is provided to the system, **x = 0100010** and **x_valid** is driven correctly.
8. The simulation ends setting **testing <= false**.

The [Figure 3.2] below shows the waveform for the test-bench:

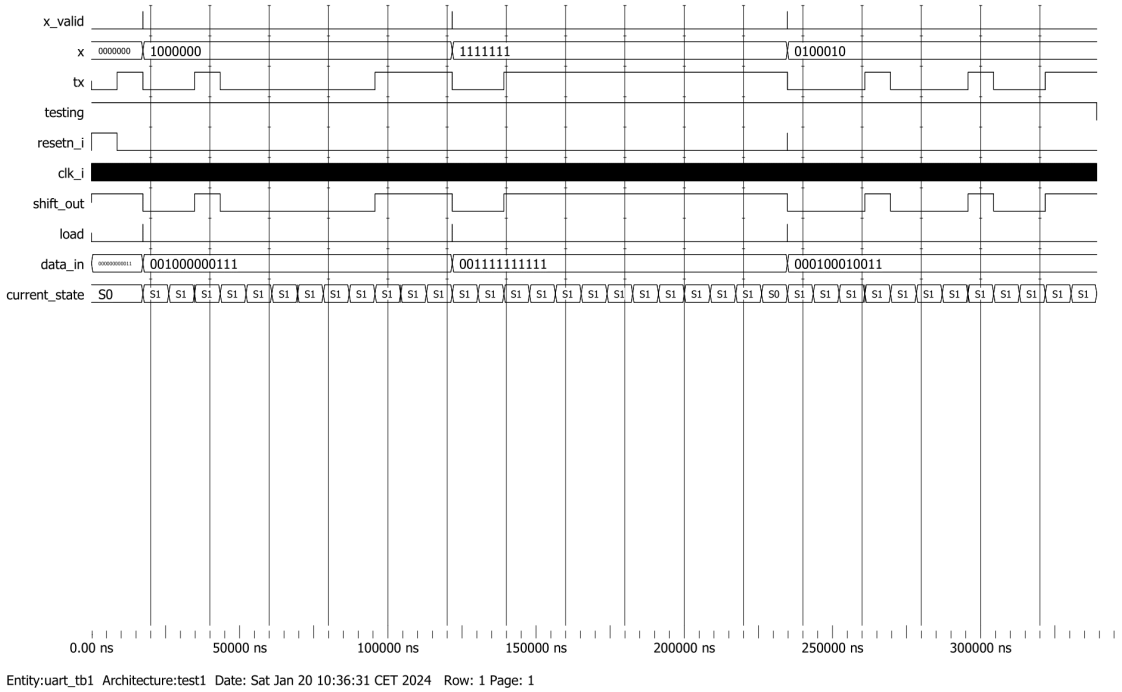


Figure 3.2: Waveform test-bench n°1.

3.5.2 Test-bench n°2

This test-bench was essential for determining whether the UART transmitter could robustly handle erroneous input drives, even when the time constraints required for a complete transmission were not met. The code, partially mentioned, is the following [Listing 3.6]:

```
STIMULUS: process
        begin
            -- Apply reset
            resetn_i <= '1';
            wait for clk_period*clk_per_bit;
            resetn_i <= '0';
            wait for clk_period*clk_per_bit;

            -- Simulating scenario where while the UART is transmitting,
            -- the input is changed and wrongfully x_valid  is set to 1.
            -- After the wrong driving, the inputs are changed again (in this case respecting the time constraint),
            -- in order to check if the trasmitter is still working.
            x_valid <= '1';
            x <= "1000000";
            wait for clk_period;
            x_valid <= '0';
            wait for clk_period*clk_per_bit*6;
            x_valid <= '1';
            x <= "0000000";
```

```
20              wait for clk_period;
21              x_valid <= '0';
22              wait for clk_period*clk_per_bit*6;
23              x_valid <= '1';
24              x <= "0101010";
25              wait for clk_period;
26              x_valid <= '0';
27              wait for clk_period*clk_per_bit*12;
28              testing <= false;
29            -- End of simulation
30            wait;
31        end process;
```

Listing 3.6: Second test-bench code.

The code provided above represents the following flow of **inputs**:

1. Firstly, a **reset** signal is applied for a total of **clk_period*clk_per_bit ns**.
2. After **clk_period*clk_per_bit ns**, the UART transmitter receives **x = 1000000** and **x_valid** is driven correctly in order to start the transmission.
3. Once elapsed **clk_period*clk_per_bit*6 ns**, the input is changed wrongfully while the system is still transmitting.
4. To ensure the UART is functioning **properly**, a **new transmission** is requested after a delay of **clk_period*clk_per_bit*6 ns**. For this transmission, all inputs are correctly driven and the system's state is **idle**. This verifies the system's **correct** behaviour.

The [Figure 3.3] below shows the waveform for the test-bench:

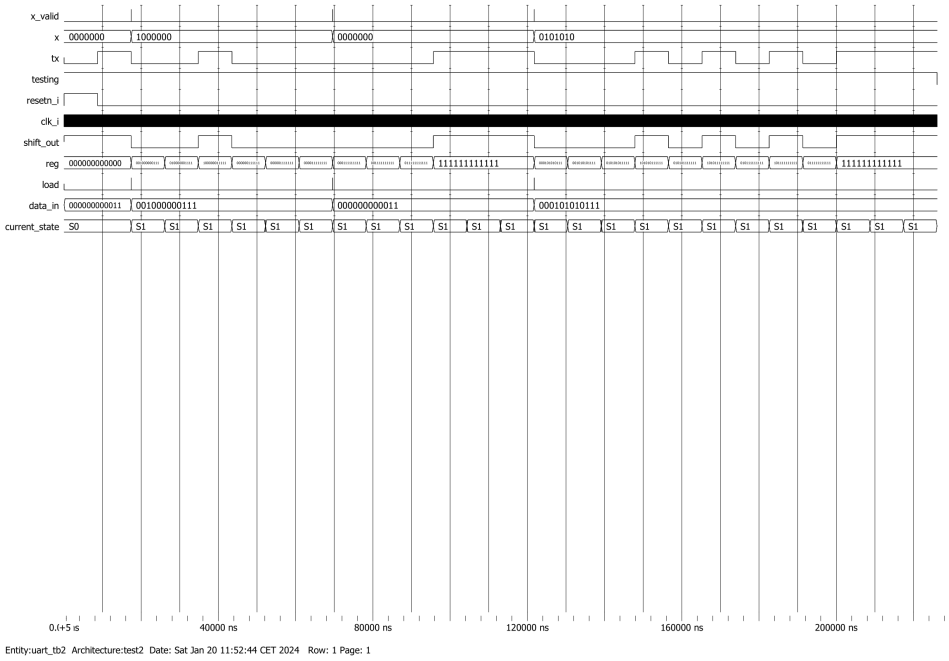


Figure 3.3: Waveform test-bench n°2.

3.5.3 Test-bench n°3

Finally, this test-bench was crucial in assessing whether the UART transmitter remains unresponsive when **x_valid** is not driven correctly. The code, partially mentioned, is the following [Listing 3.7]:

```
STIMULUS: process
        begin
            -- Apply reset
            resetn_i <= '1';
            wait for clk_period*clk_per_bit;
            resetn_i <= '0';
            wait for clk_period*clk_per_bit;

            -- Simulating a scenario where the input x
            --is changed while x_valid is not driven correctly.
            -- Then we start a new transmission with a
            -- new x and a correct x_valid.
            x_valid <= '1';
            x <= "1000000";
            wait for clk_period;
            x_valid <= '0';
            wait for clk_period*clk_per_bit*12;
            x <= "0000000";
            wait for clk_period*clk_per_bit*6;
            x_valid <= '1';
            x <= "0101010";
            wait for clk_period;
            x_valid <= '0';
            wait for clk_period*clk_per_bit*12;
            testing <= false;
           -- End of simulation
            wait;
        end process;
```

Listing 3.7: Third test-bench code.

The code provided above represents the following flow of **inputs**:

1. Firstly, a **reset** signal is applied for a total of **clk_period*clk_per_bit ns**.
2. A new transmission is started. **x = 1000000** and **x_valid** is driven correctly.
3. A new input is sent to the UART, but **x_valid** is not driven correctly.
4. The system doesn't react and goes into the **idle** state when the previous transmission is completed.
5. A new transmission is requested, **x = 0101010** and **x_valid** is driven correctly.

The [Figure 3.4] below shows the waveform for the test-bench:

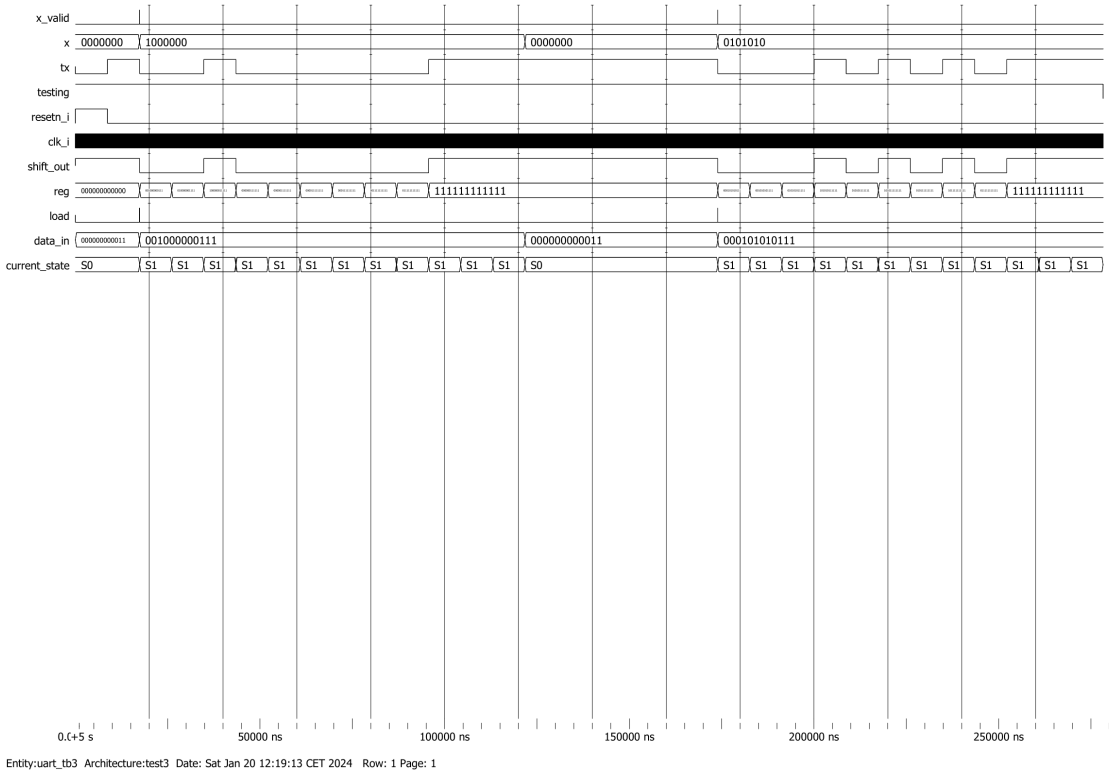


Figure 3.4: Waveform test-bench n°3.

Chapter 4

Vivado Implementation

Due to the dual architecture implemented for the parallel shift-left register, it is necessary to conduct a comparative analysis of the different designs with the objective of evaluating the one that shows superior efficiency.

4.1 First version

The first UART version is based on the code illustrated in [Listing 3.2]. Initially, the system's synthesis and implementation were executed by establishing a clock constraint equivalent to **8 ns / 125 MHz** as designed and implemented in the VHDL code. For the assignment purposes, the system was tested with a higher clock frequency. However, at a frequency of **3ns / 333,333 MHz**, the **Worst Negative Slack (WNS)** transitions into a negative state, resulting in the failure of one endpoint. The data derived from the implementation are tabulated in the subsequent [Table 4.1] and [Table 4.2].

Clock	Worst Negative Slack	Power Consumption
125 MHz	3,858 ns	0,093 W
250 Mhz	0,398 ns	0,095 W
333 Mhz	-0,042 ns	0,097 W

Table 4.1: Implementation of the first UART version with three different clock constraints.

Clock	SliceLUTs	SliceRegisters	Slice	LUTasLogic	BondedIOB
125 MHz	33	37	16	33	11
250 MHz	34	37	14	34	11
333 MHz	43	37	19	43	11

Table 4.2: Utilization of the first UART version with three different clock constraints.

4.2 Second version

The second UART version is based on the code illustrated in [Listing 3.3]. Initially, the system's synthesis and implementation were executed by establishing a clock constraint equivalent to **8 ns / 125 MHz** as designed and implemented in the VHDL code. For the objectives of the assignment, the system was subjected to testing at an elevated clock frequency. Nonetheless, at a frequency of **3 ns / 333,333 MHz**, the magnitude of the **Worst Negative Slack (WNS)** diminishes, but it does not descend below zero. The data derived from the implementation are tabulated in the subsequent [Table 4.3] and [Table 4.4].

Clock	**Worst Negative Slack**	**Power Consumption**
125 MHz	4,253 ns	0,093 W
250 Mhz	0,760 ns	0,095 W
333 Mhz	0,196 ns	0,097 W

Table 4.3: Implementation of the second UART version with three different clock constraints.

Clock	**SliceLUTs**	**SliceRegisters**	**Slice**	**LUTasLogic**	**BondedIOB**
125 MHz	25	38	12	25	11
250 MHz	26	38	12	26	11
333 MHz	38	38	17	38	11

Table 4.4: Utilization of the second UART version with three different clock constraints.

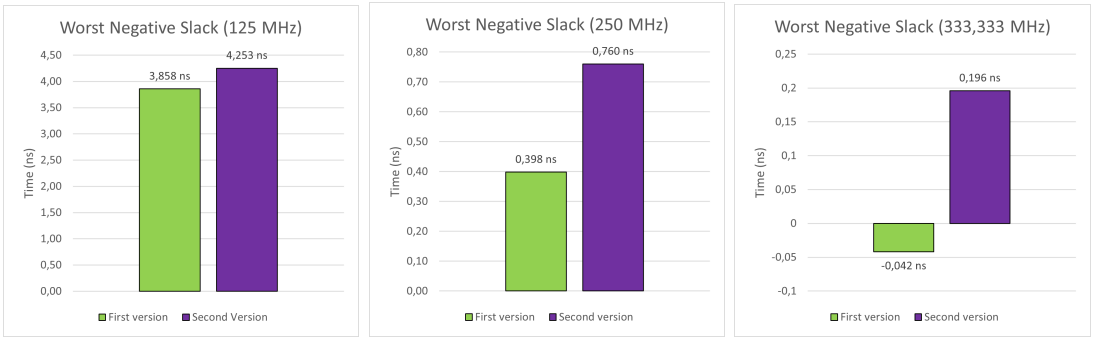
Chapter 5

Conclusions

In order to understand which version is more efficient, it can be conducted an analytical review of the data procured from the synthesis and implementation processes.

5.1 Data Analysis

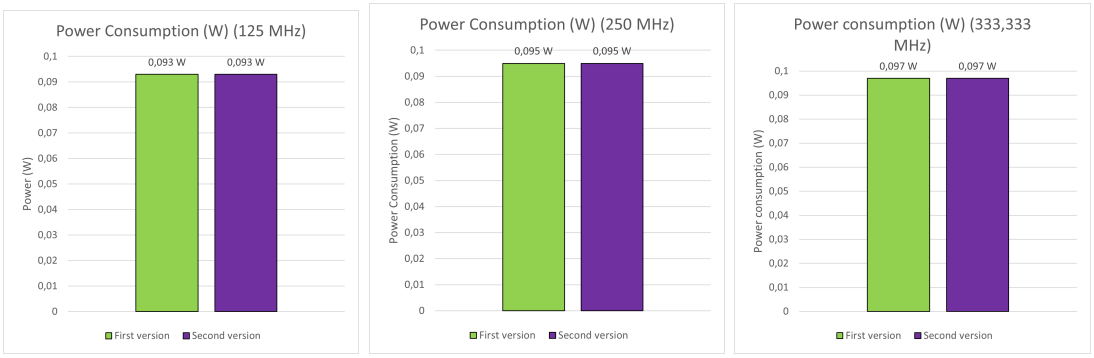
Worst Negative Slack (WNS)



(a) 125 MHz (b) 250 MHz (c) 333 MHz

Figure 5.1: Comparison between WNSs: The second version has the greatest Worst Negative Slack.

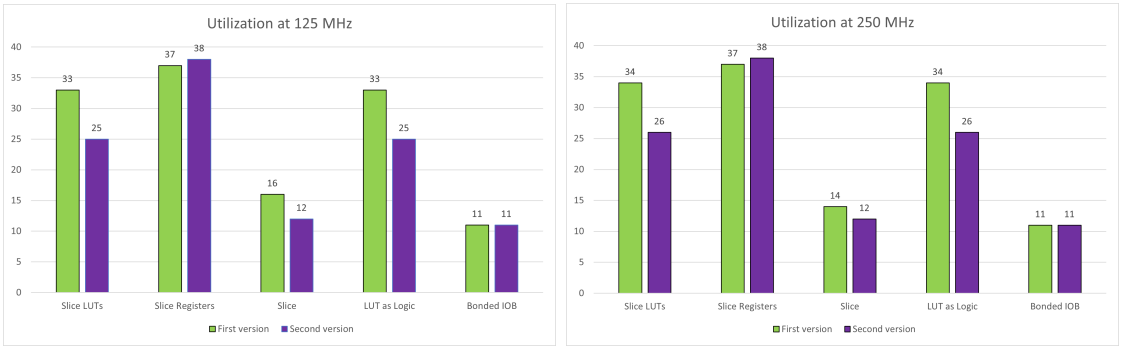
Power Consumption



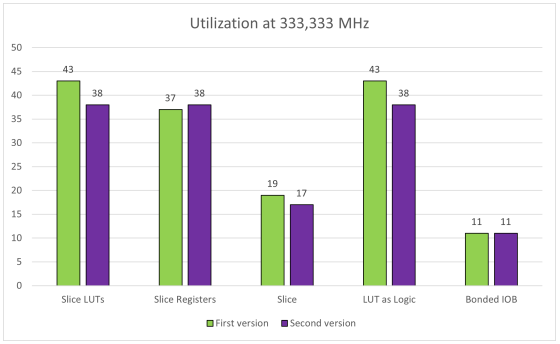
(a) 125 MHz (b) 250 MHz (c) 333 MHz

Figure 5.2: Comparison between PCs: The power consumption is observed to be analogous.

Utilization



(a) 125 MHz (b) 250 MHz



(c) 333 MHz

Figure 5.3: Comparison between utilization: it is observed that the second version demonstrates a decrease in utilization by 17,1%, 15% and 7,8% respectively.

Upon concluding the analysis, it is clear that the UART, which was implemented with the **second version** of the parallel shift left register, **exhibits a marginal efficiency superiority** over the first version:

1. It can run at **higher frequencies**, although the system was originally designed to operate **at 125 MHz**. The system's modularity allows that the **clk_per_bit** parameter can be adjusted to run at a higher frequencies.
2. At **125 MHz**, it uses **17,1%** fewer logical resources.
3. The power consumption of both designs is nearly **identical**.
4. The VHDL code is more refined and easier to read.